PRACTICUM 3

Daan Hoogland  MAARTEN PETIT

Searching and sorting

Contents

[Inleiding 2](#_Toc408775712)

[Implementatie van substring in programmeertalen 3](#_Toc408775713)

[Implementatie substring algoritme in Java 3](#_Toc408775714)

[Implementatie substring algoritme in C++ 3](#_Toc408775715)

[Source code Java methode indexOf() 4](#_Toc408775716)

[Substring Seach Algoritmes 5](#_Toc408775717)

[Knuth-Morris-Pratt 5](#_Toc408775718)

[Boyer-Moore 5](#_Toc408775719)

[Resultaten 6](#_Toc408775720)

[Reguliere Expressies 7](#_Toc408775721)

[Unittests 7](#_Toc408775722)

[Tabel met testgevallen 8](#_Toc408775723)

# Inleiding

Practicum opdracht drie van het vak Sorting & Searching bestaat uit drie onderdelen.

De eerste twee onderdelen gaan beide over substring algoritmes. We gaan van de programmeertalen Java & C++ de implementatie van het substring algoritme bekijken.

Bij het tweede onderdeel gaan we twee substring algoritmes vergelijk. De twee algoritmes die we gaan vergelijken zijn Knuth-Morris-Pratt en Boyer-Moore.

Het laatste onderdeel gaat over reguliere expressies. We gaan een methode maken in Java die controleert of een telefoonnummer voldoet aan een aantal gestelde eisen.

# Implementatie van substring in programmeertalen

In dit onderdeel gaan we bekijken hoe het substring algoritme is geïmplementeerd in de programmeertalen Java & C++. Door de sourcecode te bekijken gaan we na om welke variant van het substring algoritme het gaat.

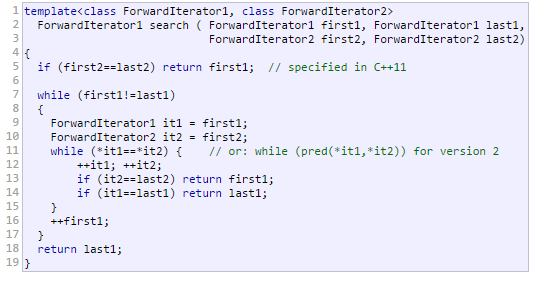
## Implementatie substring algoritme in Java

De Source code van het substring algoritme voor Java staat op de volgende pagina.

Bij de implementatie van het substring algoritme in Java gaat het om de brute force variant. De eerste letter van de meegegeven string wordt vergeleken met de eerste letter van de source string. Komt deze overeen dan wordt de tweede letter vergeleken. Komen alle letters overeen dan is de string aanwezig en returend de methode de index hiervan. Komt de letter niet overeen. Dan wordt de eerste letter vergeleken met de volgende letter in de source String. Wanneer het einde van de string wordt bereikt is de string ook niet aanwezig en returned de methode -1.

## Implementatie substring algoritme in C++

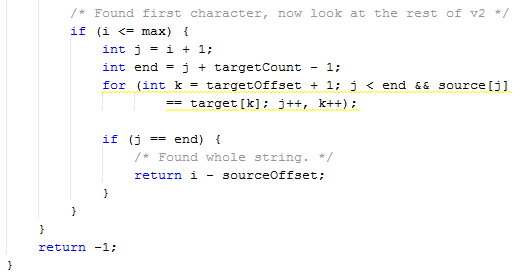
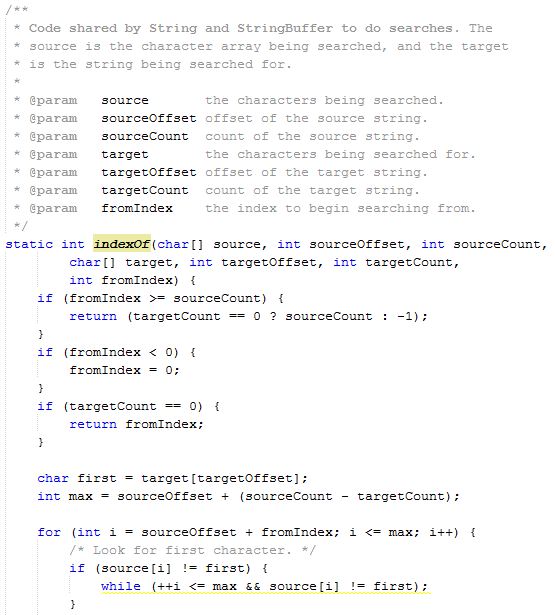
De source code van het substring algoritme in C++ (methodenaam: search)  
Return value: An iterator to the first element of the first occurrence of [first2,last2) in [first1,last1).  
If the sequence is not found, the function returns last1.



*Bron:* [*http://www.cplusplus.com/reference/algorithm/search/*](http://www.cplusplus.com/reference/algorithm/search/)

Bij de implementatie van het substring algoritme in C++ wordt ook gebruik gemaakt van de brute force methode. De methode begint bij de eerste letters van de strings. Zijn deze twee letter gelijk dan wordt de volgende letter vergeleken. Wanneer de laatste letter ook gelijk is stopt de loop en returned de methode de iterators naar de plek van de string als eerste voorkomt. Wanneer de letters niet overeenkomen gaat de methode terug naar de eerste vergeleken letter van de huidige reeks. Vanaf hier gaat methode weer opnieuw verder.

## Source code Java methode indexOf()



# Substring Seach Algoritmes

Als 2e deel van de opdracht moeten wij 2 substring search algoritmes met elkaar vergelijken. Deze 2 zijn Knuth-Morris-Pratt en Boyer-Moore. In de rest van dit hoofdstuk word naar deze 2 algoritmes verwezen met hun afkortingen, KMP en BM. Deze stukken code komen direct uit het boek, maar zijn aangepast om niet te stoppen bij het eerste woord, maar om door te gaan en de hoeveelheid worden te tellen.

## Knuth-Morris-Pratt

/\*\*

\* Searches the given text for the String pattern given in the constructor.

\*

\* **@param** txt Text to be searched for pattern.

\* **@return** Number of occurances of pattern within the String txt

\*/

**public** **int** search(String txt) {

**int** count = 0, characterCompares = 0;

**int** i, j, N = txt.length(), M = pat.length();

**for** (i = 0, j = 0; i < N && j < M; i++) {

characterCompares++;

j = dfa[txt.charAt(i)][j];

**if** (j == M) {

count++;

i++;

j = 0;

}

}

System.***out***.println("Character comparisons: " + characterCompares);

System.***out***.println("Word count: " + count);

**return** count;

}

In de bovenstaande code wordt gezocht in de String txt naar het patroon dat aangegeven is in de constructor. De count variabele houdt de hoeveelheid woorden die het algoritme tegenkomt die overeen komen met het patroon bij. Normaal zou het programma hier stoppen, maar omdat j weer naar 0 gezet word, gaat het verder.

Om het aantal vergelijkingen te tellen staat er een counter binnen de loop, boven de vergelijkingen. Hier valt niet veel over te zeggen omdat deze simpelweg het aantal vergelijkingen bijhoud dat nodig is om door de hele tekst heen te lopen.

## Boyer-Moore

/\*\*

\* Searches the given text for the String pattern given in the constructor.

\*

\* **@param** txt Text to be searched for pattern.

\* **@return** Number of occurances of pattern within the String txt

\*/

**public** **int** search(String txt) {

**int** count = 0, characterCompares = 0;

**int** M = pat.length();

**int** N = txt.length();

**int** skip;

**for** (**int** i = 0; i <= N - M; i += skip) {

skip = 0;

**for** (**int** j = M - 1; j >= 0; j--) {

characterCompares++;

**if** (pat.charAt(j) != txt.charAt(i + j)) {

skip = Math.*max*(1, j - right[txt.charAt(i + j)]);

**break**;

}

}

**if** (skip == 0) {

count++;

skip = 1;

}

}

System.***out***.println("Character comparisons: " + characterCompares);

System.***out***.println("Word count: " + count);

**return** count;

}

Net zoals bij het KMP algoritme hierboven moest de code worden aangepast om de woorden en het aantal vergelijkingen te tellen. Hierbij worden gebruik gemaakt van dezelfde 2 variabelen.

Als skip gelijk is aan 0 betekend dat er een match gevonden is, waardoor de count variabelen 1 opgehoogd moet worden.

Bij de characterCompare variabele moet elke keer als er een karakter vergeleken word 1 erbij komen. Deze staat in de binnenste for loop. Deze variabele zou hierdoor dus minder voorkomen omdat de buitenste for loop veel grotere stappen maakt.

## Resultaten

Voor de resultaten hebben wij 10 woorden genomen en gecheckt hoe vaak deze voorkwamen in de applicatie. Daarnaast hebben wij bijgehouden hoeveel vergelijkingen elk algoritme doet om bij dit nummer te komen. Hieronder in de tabel is dit te zien:



Wat al duidelijk was is dat de aantallen altijd gelijk zijn, maar de vergelijkingen verschillen enorm van elkaar. Bij KMP zijn de vergelijkingen ongeveer altijd gelijk, maar bij BM verschillen ze erg.

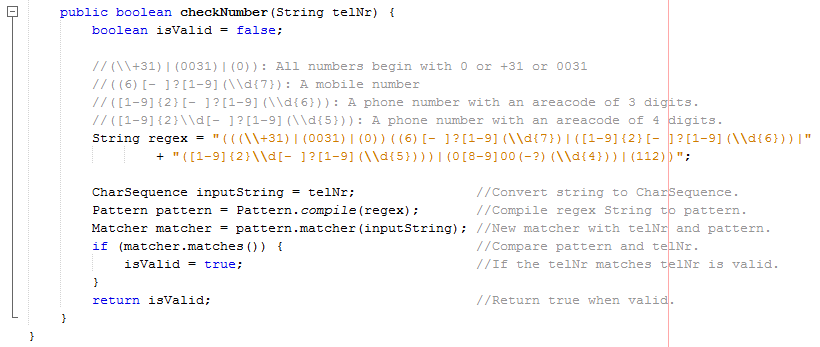
De belangrijkste reden waarom BM een kleiner aantal vergelijkingen heeft is omdat BM beter werkt op grotere lappen tekst en bij langere woorden efficiënter is (wat ook te zien is in de resultaten). Dit komt omdat als er een letter vergeleken word die helemaal niet in het patroon zit, het de gehele lengte van het word overslaat en vanaf daar verder gaat kijken.

Bij KMP vergelijkt het steeds elk karakter in de zoekstring met het patroon, waardoor hij elke keer bijna de hele tekst door moet lopen.

Onze resultaten komen ongeveer overeen met de resultatentabel uit het boek (pagina 779)

# Reguliere Expressies

Bij dit onderdeel van het practicum gaan we een reguliere expressies maken om te testen of een telefoonnummer aan een aantal gestelde eisen voldoet. Om een telefoonnummer te checken hebben we de methode checkNummer(String telNr) gemaakt.

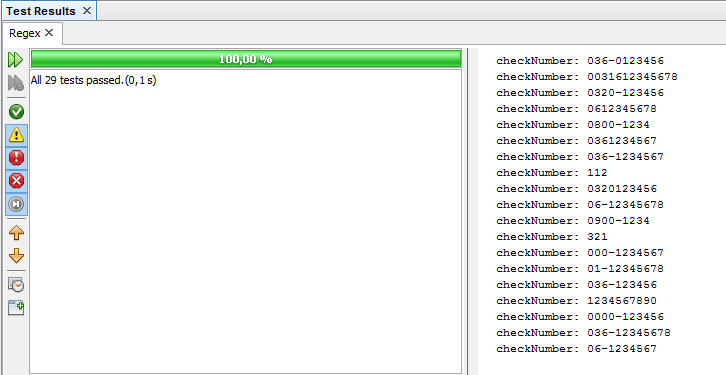


De reguliere expressie: “(((\\+31)|(0031)|(0))((6)[- ]?[1-9](\\d{7})|([1-9]{2}[- ]?[1-9](\\d{6}))|([1-9]{2}\\d[- ]?[1-9](\\d{5})))|(0[8-9]00(-?)(\\d{4}))|(112))”.

## Unittests

Om de methode te testen hebben we een aantal unittests gemaakt. Op de volgende pagina staat een tabel met alle testgevallen, de correcte uitkomst en de reden waarom we het nummer hebben getest.

Output van de unittests:



## Tabel met testgevallen

|  |  |  |
| --- | --- | --- |
| **Telefoonnummer:** | **Correct uitkomst:** | **Reden** |
| 06-12345678 | True | Mobiel nummer met streepje moet valid zijn. |
| 0612345678 | True | Mobiel nummer zonder streepje moet valid zijn. |
| 036-1234567 | True | Telefoonnummer met 3-cijferig netnummer met streepje moet valid zijn. |
| 0361234567 | True | Telefoonnummer met 3-cijferig netnummer zonder streepje moet valid zijn. |
| 0320-123456 | True | Telefoonnummer met 4-cijferig netnummer met streepje moet valid zijn. |
| 0320123456 | True | Telefoonnummer met 4-cijferig netnummer zonder streepje moet valid zijn. |
| 112 | True | 112 is een uitzondering en moet valid zijn. |
| 0900-1234 | True | 0900 nummers met streepje moeten valid zijn. |
| 0800-1234 | True | 0800 nummers met streepje moeten valid zijn. |
| 09001234 | True | 0900 nummers zonder streepje moeten valid zijn. |
| 08001234 | True | 0900 nummers met streepje moeten valid zijn. |
| +31361234567 | True | 0800 nummers met streepje moeten valid zijn. |
| 0031361234567 | True | Telefoonnummer met landcode moet valid zijn. |
| +31612345678 | True | Mobiel nummer met landcode moet valid zijn. |
| 0031612345678 | True | Mobiel nummer met landcode moet valid zijn. |
|  |  |  |
| 06-01234567 | False | Abonneenummer mag nooit met een 0 beginnen. |
| 036-0123456 | False | Abonneenummer mag nooit met een 0 beginnen. |
| 0320-012345 | False | Abonneenummer mag nooit met een 0 beginnen. |
| 0601234567 | False | Abonneenummer mag nooit met een 0 beginnen. |
| 0320012345 | False | Abonneenummer mag nooit met een 0 beginnen. |
| 06-123456789 | False | Een nummer heeft 10 cijfers |
| 06-1234567 | False | Een nummer heeft 10 cijfers |
| 036-12345678 | False | Een nummer heeft 10 cijfers |
| 036-123456 | False | Een nummer heeft 10 cijfers |
| 1234567890 | False | Een nummer begint altijd met 0, +31 of 0031 |
| 0000-123456 | False | Een netnummer heeft nooit een 0 als 2e of 3e getal. |
| 000-1234567 | False | Een netnummer heeft nooit een 0 als 2e of 3e getal. |
| 01-12345678 | False | Een mobielnummer heeft altijd als netnummer 06 |
| 321 | False | 0900 nummers met streepje moeten valid zijn. |